Week#7 RocksDB Compaction

Sangeun Chae

2018314760

# INTRODUCTION

RocksDB는 Facebook에서 시작된 오프소스 데이터베이스 개발 프로젝트로, 서버 워크로드와 같은 대용량 데이터처리에 적합하고 빠른 저장장치, 특히 플래시 저장장치(SSD)에서 높은 성능을 내도록 최적화되어 있다. RocksDB에는 크게 3가지 핵심 요소가 있다. Memtable, SST files, WAL. 이번 랩에서는 SST files간에 일어나는 compaction과정을 다뤄보고자 한다. Memtable은 Memory 내부에 전재하며 disk의 SST file로 Flush되기 전에 데이터를 저장하는 데이터 구조이다. 쓰기 요청이 들어오면 반드시 memtable에 우선적으로 쓰이며, 읽기 또한 RocksDB의 특성상 가장 최신의 데이터부터 찾기 시작하므로 memtable부터 확인한다. 만약 memtable이 특정 threshold 크기 이상이 되면, 데이터는 변경이 불가능한 읽기 전용 (Immutable memtable)로 flush됩니다. 만약 읽기 전용 memtable도 가득 차면, disk로 flush가 된다. Memtable 이 flush되게 되면, 가장 먼저 level0의 SST file로 가게 된다. SST file은 Key의 순서에 따라 정렬이 되어 있는데, Level 0는 각 SST file마다 Key 값이 겹칠 수 있다. 만약 Level 0의 크기가 특정 threshol를 넘어가게 되면, 2가지 compaction policy를 선택하여 compaction을 진행할 수 있다. 이번 랩에서는, 두가지 compaction policy에 대해서 다뤄볼 예정이다.

# METHODS

RocksDB DB\_bench를 실행시킬 때, compaction style을 leveled compaction(compaction style:0), universal compaction(compaction style: 1) 로 parameter를 변경하면서 번갈아 실행한다. 각각의 DB\_bench를 실행 후, LOG 파일을 상위 디렉토리에 복사를 하고(복사를 하지 않으면, log가 append되기 때문에 기존의 log파일이 삭제된다), 2가지의 bench가 끝나면 두개의 log 파일을 비교한다

.

# Performance Evaluation

## Experimental Setup

|  |  |
| --- | --- |
| **Type** | **Specification** |
| OS | Ubuntu 20.04.3 LTS |
| CPU | AMD Ryzen 7 5800X 8-Core Processor (VMware support 4 Core) |
| Memory | 4GB |
| Kernel | Linux ubuntu 5.11.0.34 -generic |
| Disk | VMware Virtual 80GB |

Table : System setup

|  |  |
| --- | --- |
| **Type** | **Configuration** |
| Bench Type | “readrandomwriterandom” |
| Direct flush\_compaction | True |
| Direct read | True |
| Duration | 7200s |

Table : Benchmark setup

## Experimental Results

우선, compaction policy를 leveled compaction policy를 사용했을 때의 결과이다.

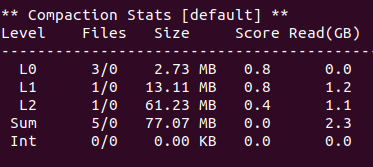


Figure : Leveled Compaction [1]

텍스트, 검은색, 계산기, 다른이(가) 표시된 사진

자동 생성된 설명

Figure : Leveled Compaction [2]

테이블이(가) 표시된 사진

자동 생성된 설명

Figure : Leveled Compaction [3]

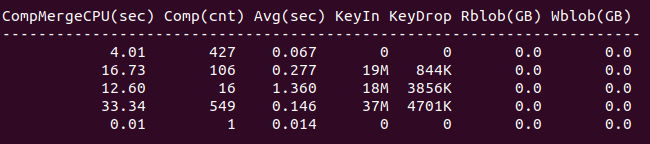


Figure : Leveled Compaction [4]

Leveled compaction은 L0의 파일 수가 특정 threshold에 도달했을 때 발생하고, L0의 파일들은 L1에 병합(Merge)된다. L0의 파일들은 파일별로 키가 overlapping되기 때문에, 모든 L0에 존재하는 파일들은 compaction에 참여하게 된다. 따라서 L0의 파일들을 L1과 merge하고 나면, L1 level에서도 threshold를 초과하게 되는 상황이 올 수 있다. 이러한 경우에는, L0 level과 다르게 특정 파일만 L2 level에 compaction되는 과정을 거치게 된다. 따라서 level이 증가할수록, 오래된 데이터가 축적되는 형태가 만들어 지게 된다.

다음은, compaction policy를 universal compaction policy (tiered compaction policy)를 사용했을 때의 결과이다.

텍스트, 검은색, 전자기기이(가) 표시된 사진

자동 생성된 설명

Figure : Universal Compactions [1]

텍스트, 원격, 검은색, 다른이(가) 표시된 사진

자동 생성된 설명

Figure : Universal Compactions [2]

텍스트, 전자기기이(가) 표시된 사진

자동 생성된 설명

Figure : Universal Compactions [3]

텍스트, 전자기기이(가) 표시된 사진

자동 생성된 설명

Figure : Universal Compactions [4]

Universal compaction은 Leveled compaction과 다르게, 파일 수가 threshold에 도달했을 때 (space threshold가 도달했을 때) compaction되는 것은 동일하지만, overlapping되는 키가 각각의 level별로 존재한다. 즉, 시간을 기점으로 compaction을 하기 때문에, sorted run에는 overlapping되는 키가 존재하지 않지만, level별로는 존재하게 된다.

# Conclusion

Leveled compaction을 수행하게 되면, 사용되는 space의 크기는 overlapping되는 key가 없기 때문에, 많은 storge를 사용하지 않게 된다. 또한 read하는 과정에 있어서, overlapping되는 키가 없기 때문에, 각각의 level에서 특정 key가 존재하는 파일만 search하면 되기 때문에, read에 소요되는 overhead가 줄어든다. 하자만, compaction과정에 있어서 같은 key가 여러 번 write될 수 있기 때문에, write amplification이 발생하게 되고, 그로 인해 write stall이 발생하여 write performance의 저하를 가져온다. 반면, universal compaction 은 level별로 overlapping되는 key가 존재하기 때문에, 전반적인 space는 늘어나게 된다. 이 과정에서 space amplification이 발생하게 된다. 또한 read query를 수행하는 것에 있어서도, overlapping된 key를 모두 search해야하기 때문에, read performance의 저하를 가져온다. 하지만 write에 있어서는, write amplification이 leveled compaction에 비해 작기 때문에, write performance는 좋아짐을 알 수 있다. 따라서 heavy write workload 환경에서는 leveled compaction보다 universal compaction을 수행하는 것이 performance과점에서는 유리하다.

# REFERENCES

1. <https://meeeejin.gitbooks.io/rocksdb-wiki-kr/content/leveled-compaction.html>
2. <https://github.com/facebook/rocksdb/wiki/Universal-Compaction>